Final Project Report

Comparison of AI Search Algorithms for Solving the Sliding Puzzle Problem

Mohammed Farhan Baluch [110093799]

Yameen Ajani [110096721]

January 6, 2023

Contents

Abstract

When dealing with problems in Artificial Intelligence, one of the most common situations that arise is finding a path from the current state to a specified goal state. To make the solution efficient, we are always concerned with finding the shortest path to the goal state. Over the years, several search algorithms have been introduced, each with its unique way of providing a solution. Moreover, each method has its advantages and disadvantages, because of which they tend to work better on a particular type of problem. However, it is essential to study the utilization of resources by a particular algorithm for it to be used in practice. In this project, we have compared some of the widely used Artificial Intelligence search algorithms for the sliding puzzle problem based on parameters like time taken and memory usage.

1 Introduction

The sliding puzzle problem is a classic AI problem that involves moving a set of tiles on a board to a goal configuration. It is often used as a benchmark for testing the performance of AI algorithms, as it requires a combination of search and heuristics to solve efficiently.

The puzzle consists of a grid of tiles, with one tile missing. The player can slide any tile that is adjacent to the empty space into the empty space, with the goal of arranging the tiles in a specific order. This problem can be represented as a state space search problem, where the initial state is the starting configuration of the tiles and the goal state is the solved configuration.

The Eight and Fifteen Puzzle, among other sliding-tile puzzles, have long been used as testbeds for heuristic search in Al. Numbered tiles are arranged in a square frame, with the so-called "blank" slot remaining unfilled. You can slide any tile that is either horizontally or vertically adjacent to the blank into that location. The purpose is to rearrange the tiles from a given goal configuration into an ideal or optimal configuration in the fewest possible movements. There are more than 10^5 nodes in the state space for the Eight Puzzle, 10^{13} nodes for the Fifteen Puzzle, and over 10^{25} nodes for the Twenty-Four Puzzle.

In recent years, the sliding puzzle problem has received renewed interest as a testbed for artificial intelligence and machine learning techniques. Researchers have explored the use of deep learning and reinforcement learning to learn heuristics or even solve the puzzle directly from raw image inputs.

Overall, the sliding puzzle problem remains an important and challenging problem in AI research, with the potential to advance our understanding of search and heuristics, as well as machine learning and intelligent agents.

In this paper, we perform a comparative analysis of all classic AI algorithms and find the best performing algorithm to solve the randomly generated 4x4 sliding puzzle problem.

2 Literature Review

2.1 Relevant History

The sliding puzzle problem, also known as the 15 puzzle, is a popular puzzle that has been around since the late 19th century. It consists of a frame of numbered square tiles in random order with one tile missing. The object of the puzzle is to arrange the tiles in numerical order by sliding them around within the frame.

The 15 puzzle was invented in the 1870s by Noyes Chapman, a postmaster in Canastota, New York. It was originally called the "14-15 Puzzle," because it had 14 numbered tiles and one blank space. The puzzle became very popular in the United States in the late 1800s and early 1900s, and it was sold in various sizes and shapes, including circular and star-shaped versions.

The sliding puzzle problem has a long history, dating back to the late 19th century, and has been a popular subject of study in the field of artificial intelligence (AI). AI algorithms are designed to enable computers to solve problems in a way that is similar to how humans solve problems, and the sliding puzzle problem is an excellent example of a problem that can be solved using AI.

One of the earliest AI algorithms to be developed for solving the sliding puz-

zle problem was the A* algorithm, which was developed in the 1960s. The A* algorithm is a type of search algorithm that is used to find the shortest path between two points, and it has been used extensively to solve the sliding puzzle problem. The Eight Puzzle's ideal solutions may be discovered via a breadth-first search because of the puzzle's constrained search space. Using Iterative Deepening-A* (IDA*) and the Manhattan distance heuristic function, Korf initially discovered the best answers to the Fifteen Puzzle [1].

2.2 Recent Works

Korf et. Al. proposed a powerful heuristic function to tackle the randomly generated 10 instances of twenty-four puzzle problem. They implemented IDA* with this novel heuristic and a method for pruning duplicate nodes in DFS. Results showed average optimal solution length of 102.6 moves [2].

Dobbelin et. Al. presents a parallel algorithm that performs BFS in the compressed pattern space thereby computing very large pattern database(PDB) heuristics. They built compressed version of 3 large PDBs and implemented the algorithm on Korf's set of random 24-puzzle instances. Their experiments indicate an average 8-fold improvement of the 9-9-6 PDB over the 6-6-6-6 PDB on the 24-puzzle. Combining several large PDBs yields a 13-fold improvement [3].

Korf proposed a scalable implementation of Delayed Duplicate Detection(DDD) with several AI algorithms. He implemented best-first frontier search with DDD and A* with DDD (without frontier) on the 15-puzzle problem. A* with DDD can store 70 million nodes in the same amount of memory as standard A* can store about 35 million, and can solve 87/100 problems, but runs about three times slower than standard A*. Breadth- first frontier search required 19.75 minutes, and breadth-first frontier search with DDD required 9.25 minutes, on a 440 Megahertz Sun Ultra 10 workstation. This shows that DDD can be useful even for problems that fit in memory [4].

A* typically returns solutions that are better than a given bound requires. Author shows how to take advantage of this behavior to speed up search while retaining bounded suboptimality. Thayer et. Al. present an optimistic algorithm that uses a weight higher than the user's bound and then attempts to prove that the resulting solution adheres to the bound. They tested the

algorithm on the 100 benchmark 15-puzzle instances from Korf (1985) using the standard Manhattan distance heuristic. Optimistic search appears to be outperforming the other algorithms, and once again it appears to behave as if it were weighted A*, run with a higher weight [5].

Tuncar et. al. proposes a novel method for solving 15-puzzle problem by using Artificial bee colony algorithm. They use linear conflict function with the Manhattan distance heuristic's to increase its effectiveness, while the pattern database technique was used to find solutions more quickly. When tested on randomly generated 25 problems, the results were equivalent and, in some circumstances, slightly better than IDA*. [6].

For large-scale sliding puzzles, Wang et. al. introduces a DSolving method that provides an innovative and efficient solution. It does not need the storing of intermediate states and hence shows the potential to solve any scale challenge. It employs an effective way to transport most number tiles to their intended places via the shortest paths. DSolving creates state transition tables for certain sub-puzzles to simplify the issue to simple look-up procedures. On a home computer, this approach results to be very time-efficient and reliable, requiring only 4-5 milliseconds to complete a 20x20 problem [7].

3 Chosen Methods

There are several AI search algorithms that can, given a problem, help find the shortest path to the goal state.
The algorithms we have used in this comparative study are discussed as follows.

3.1 Breadth First Search (BFS)

Breadth-first search (BFS) is an algorithm that traverses a graph or tree data structure breadth-first, meaning that it explores all the neighbor nodes at the present depth level before moving on to the nodes at the next depth level. It is often used to find the shortest path between two nodes in a graph or tree.

To perform a BFS, we start at a designated root node and explore all of its

neighbors before moving on to any of their neighbors. We keep track of the nodes we have already visited in a queue, which allows us to ensure that we visit each node exactly once and maintain the breadth-first order.

One way to implement BFS is to use a queue to store the nodes that are waiting to be explored. We start by enqueuing the root node and marking it as visited. Then, we dequeue a node from the front of the queue and explore all of its neighbors. If a neighbor has not been visited, we mark it as visited and enqueue it. This process continues until the queue is empty, at which point we have fully traversed the graph or tree. This makes it relatively efficient for small graphs, but it may not be suitable for very large graphs with millions of nodes.

3.2 Depth First Search (DFS)

Depth-first search (DFS) is an algorithm that traverses a graph or tree data structure depth-first, meaning that it explores as far as possible along each branch before backtracking. It is often used to search for a specific node or value in a graph or tree, or to check whether a path exists between two nodes.

To perform a DFS, we start at a designated root node and explore as far as possible along each branch before backtracking. We keep track of the nodes we have already visited in a stack, which allows us to revisit them and explore their unvisited neighbors when we backtrack.

One way to implement DFS is to use a stack to store the nodes that are waiting to be explored. We start by pushing the root node onto the stack and marking it as visited. Then, we pop a node from the top of the stack and explore one of its unvisited neighbors. If a neighbor has not been visited, we mark it as visited and push it onto the stack. This process continues until the stack is empty, at which point we have fully traversed the graph or tree.

This makes it relatively efficient for small graphs, but it may not be suitable for very large graphs with millions of nodes. One of the main advantages of DFS is that it allows us to visit every node in the graph, even if it is not connected to the root node, which can be useful in certain applications.

3.3 Iterative Deepening Search (IDS) - Recursive and Non-recursive

Recursive Iterative Deepening Search (RIDS) is a search algorithm that combines the depth-first search (DFS) and iterative deepening search (IDS) algorithms to perform a depth-first search with increasing depth limits. RIDS performs a DFS with a depth limit that is incremented at each recursive call. If the goal is not found at a given depth limit, the limit is increased and the search is repeated until the goal is found or the maximum depth is reached. However, RIDS generally requires fewer recursive calls than IDS, which reduces the space overhead and makes it more efficient in practice.

One advantage of RIDS is that it avoids the time and space overhead of IDS while maintaining its completeness and optimality. It is a useful algorithm when the depth of the goal node is not known in advance and it is not practical to perform a full DFS or IDS. It can also be used as a fallback option when IDS fails to find the goal node within a reasonable time or space limit.

One disadvantage of IDS is that it can be slow for very large graphs or trees with a high branching factor. To improve the efficiency of IDS, it can be implemented non-recursively using a stack or queue to store the nodes that are waiting to be explored. This can reduce the space overhead and improve the performance of the algorithm.

3.4 Uniform Cost Search

Uniform Cost Search (UCS) is an algorithm that searches for a path between two nodes in a graph or tree by expanding the node with the lowest cost first. It is a variant of breadth-first search (BFS) that prioritizes nodes with a lower cost, allowing it to find the shortest path between the two nodes.

To perform a uniform cost search, we use a priority queue to store the nodes that are waiting to be explored. The priority of each node is determined by its cost, which is the total cost of the path from the start node to the current node. At each step, we dequeue the node with the lowest cost and explore all of its neighbors. If a neighbor has not been visited, we update its cost and enqueue it. This process continues until the goal node is reached or the queue is empty, at which point we have either found a path or determined that no

path exists. However, UCS generally requires fewer nodes to be expanded than BFS, as it avoids expanding nodes with a higher cost.

Overall, UCS is a useful algorithm when the cost of each step is known and we want to find the path with the lowest total cost. It is commonly used in pathfinding and robotics applications, where the cost of each step may be determined by factors such as distance, time, or energy.

3.5 A* search

A* Search (A-star Search) is an algorithm that searches for a path between two nodes in a graph or tree by expanding the node that has the lowest cost-to-go estimate. It is a variant of uniform cost search (UCS) that uses an estimate of the remaining cost to the goal node to guide the search and improve efficiency.

To perform an A* search, we use a priority queue to store the nodes that are waiting to be explored. The priority of each node is determined by the sum of its cost-to-come (the cost of the path from the start node to the current node) and its cost-to-go estimate (an estimate of the remaining cost to the goal node). At each step, we dequeue the node with the lowest total cost and explore all of its neighbors. If a neighbor has not been visited, we update its cost and enqueue it. This process continues until the goal node is reached or the queue is empty, at which point we have either found a path or determined that no path exists. However, A* search generally requires fewer nodes to be expanded than BFS, as it avoids expanding nodes with a higher cost-to-go estimate.

4 Experiments

While performing experiments, we have considered the 15-puzzle problem i.e. a 4×4 puzzle. All experiments were run on an Intel i5 processor with dual core. The system had 8 GB memory. The code was run on a macOS with Python version 3.11.0.
We implemented the above mentioned algorithms in Python and ran 10 instances of each algorithm with randomly generated initial states. For Iterative Deepening Search, we have considered a recursive as well as a non-

recursive approach. Also, for the A* Search, we have used two different heuristics - Manhattan Distance and Displaced Tiles. The two main parameters of comparison that we have considered are CPU time and memory usage. The average of the 10 instances run for every algorithm was considered and is presented in the next section.

5 Results and Discussion

In any practical environment, the two most important parameters used for analysis of any algorithm is the CPU time taken and the amount of memory used by the algorithm. Hence, we have done the same and evaluated all our algorithms on these two parameters. As mentioned in the previous section, the results are a mean of 10 random instances run on each of the algorithms. Table 1 shows the uninformed search algorithms' results. Table 2 shows

Algorithm	**CPU Time (ms)**	**Memory Usage (MB)**
Breadth First Search	641.989	12.910
Depth First Search	829.917	22.749
Recursive Iterative Deepening Search	610.187	12.478
Non-recursive Iterative Deepening Search	407.683	12.371
Uniform Cost Search	1250.976	22.996

Table 1: Results of Uninformed Search Algorithms

the results for informed search algorithms. Since the A* algorithm uses a hashmap, we use its size to evaluate the memory used by the algorithm.

It should be noted that for any sliding puzzle problem, it has been proven

Algorithm	**CPU Time (ms)**	**Hashmap Size (Bytes)**
A* Search (Manhattan Distance)	2.863	832
A* Search (Displaced Tiles)	13.128	3328

Table 2: Results of Informed Search Algorithms

that only half of the possible starting states result in a solution. Even then,

some instances are so difficult that they take a large amount of memory and hours to solve. Also, the results obtained by running these algorithms can vary based on the systems on which they are run and the resources available. Therefore, we analyze these algorithms more generally in terms of the Big O notation. Table 3 shows the time and space complexity of each of the algorithms used. The terms used in Table 3 can be interpreted as -
b - branching factor
d - depth of shortest solution
m - max depth of search tree
C - cost of optimal solution
ϵ - minimum cost of every action

Algorithm	**Time Complexity**	**Space Complexity**
Breadth First Search	$O(b^d)$	$O(b^d)$
Depth First Search	$O(b^m)$	$O(bm)$
Iterative Deepening Search	$O(b^d)$	$O(bd)$
Uniform Cost Search	$O(b^{1+C/\epsilon})$	$O(b^{1+C/\epsilon})$
A* Search	$O(b^d)$	$O(b^d)$

Table 3: Complexity of Search Algorithms

6 Conclusion

Finding a route from the present state to a designated objective state is one of the most frequent situations while solving Artificial Intelligence challenges. We are always looking for the quickest route to the desired state in order to make the solution as efficient as possible. Numerous search algorithms have been developed throughout the years, each with a distinctive approach to offering a result. Additionally, each approach has pros and cons, which is why it tends to perform best for a certain kind of problem. But before using a certain algorithm in practise, it is crucial to research how it uses resources. We have shown a comparative analysis of time complexity, space complexity, CPU Time & memory usage needed to solve the 10 randomly generated 15-puzzle instances using various classic AI algorithms.

7 Future Works

There are many different approaches that AI algorithms can take to solve the sliding puzzle problem, and in the future, it is likely that even more advanced and sophisticated techniques will be developed.
One possible example could be a neural network could be trained to take in an image of the puzzle as input and predict the best move to make next. This approach has the potential to be much faster than search algorithms, but it may require a large amount of training data and may not always find the optimal solution.
Another scope is to convert the sliding puzzle problem to boolean satisfiability problem and then feed it into a SAT solver to obtain the result. A tremendous amount of work over the past many decades has resulted in SAT solvers that are not only useful for many issues but also the fastest method now available for addressing a wide range of problems.

References

[1] Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. Artificial intelligence, 27(1), 97-109.

[2] Korf, R. E., & Taylor, L. A. (1996, August). Finding optimal solutions to the twenty-four puzzle. In Proceedings of the national conference on artificial intelligence (pp. 1202-1207).

[3] Döbbelin, R., Schütt, T., & Reinefeld, A. (2013, August). Building large compressed PDBs for the sliding tile puzzle. In Workshop on Computer Games (pp. 16-27). Springer, Cham.

[4] Korf, R. E. (2004, July). Best-first frontier search with delayed duplicate detection. In AAAI (Vol. 4, pp. 650-657).

[5] Thayer, J. T., & Ruml, W. (2008, June). Faster than Weighted A*: An Optimistic Approach to Bounded Suboptimal Search. In ICAPS (pp. 355-362).

[6] Tuncer, A. (2021). 15-Puzzle Problem Solving with the Artificial Bee Colony Algorithm Based on Pattern Database. J. Univers. Comput. Sci., 27(6), 635-645.

[7] Wang, G., & Li, R. (2017). DSolving: a novel and efficient intelligent algorithm for large-scale sliding puzzles. Journal of Experimental & Theoretical Artificial Intelligence, 29(4), 809-822.

Appendix

Individual Contribution

All the participants contributed equally to this project in all the sections including coding.